Formal Methods in System Design 8, 153–188 (1996)

Applying Formal Verification to the AAMP5 Microprocessor: *A Case Study in the Industrial Use of Formal Methods*[*]

MANDAYAM K. SRIVAS srivas@csl.sri.com
Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

STEVEN P. MILLER spmiller@cca.rockwell.com
Collins Commercial Avionics, Rockwell International, Cedar Rapids, IA 52498, USA

Abstract. Formal specification combined with mechanical verification is a promising approach for achieving the extremely high levels of assurance required of safety-critical digital systems. However, many questions remain regarding their use in practice: Can these techniques scale up to industrial systems, where are they likely to be useful, and how should industry go about incorporating them into practice? This paper discusses a project undertaken to answer some of these questions, the formal verification of the microcode in the AAMP5 microprocessor. This project consisted of formally specifying in the PVS language a Rockwell proprietary microprocessor at both the instruction-set and register-transfer levels and using the PVS theorem prover to show the microcode correctly implemented the instruction-level specification for a representative subset of instructions. Notable aspects of this project include the use of a formal specification language by practicing hardware and software engineers, the integration of traditional inspections with formal specifications, and the use of a mechanical theorem prover to verify a portion of a commercial, pipelined microprocessor that was not explicitly designed for formal verification.

Keywords: formal methods, formal verification, microprocessor verification, microcode verification, hardware verification systems, safety critical systems, PVS

1. Introduction

It is becoming increasingly evident within the VLSI design industry that the complexity of hardware designs is outstripping the capability of traditional simulation-based tools to adequately verify them. This was illustrated by the recent floating point bug discovered in Intel's Pentium processor [13]. As a result, industry is beginning to look at formal verification as a means for obtaining higher assurance than is currently possible through simulation.

Recently, SRI International and Collins Commercial Avionics, a division of Rockwell International, undertook a project to explore how formal techniques for specification and verification could be introduced into an industrial process [20]. This project, sponsored by the Systems Validation Branch of NASA Langley and Collins Commercial Avionics, consisted of specifying in the PVS language [21] a portion of a Rockwell proprietary microprocessor, the AAMP5, at both the instruction set and register-transfer levels and using the PVS interactive proof-checker [26] to show that the microcode correctly implemented the specified behavior for a representative subset of instructions.

*This work was supported by NASA Langley Research Center under contract NAS1-20334 and NAS1-19704.

The main goal of the project was two-fold. First, to investigate the feasibility of formally specifying and verifying a complex commercial microprocessor that was not expressly designed for formal verification. Second, to explore effective ways to transfer the technology to an industrial setting. The choice of the AAMP5 satisfied the first goal since the AAMP5 was not designed for formal verification, but to provide a more than threefold performance improvement while remaining object-code-compatible with the earlier AAMP2. The AAMP2 is used in numerous avionics applications, including the Boeing 737, 747, 757, and 767 aircraft.

To satisfy the technology transfer objective, we had to develop a formal infrastructure and verification methodology usable by practicing engineers. The infrastructure includes techniques for decomposing the microprocessor verification problem into a set of verification conditions that the engineers can formulate and strategies to automate the proof of the verification conditions. The development of the infrastructure was one of the key accomplishments of the project. Most of the infrastructure and methodology are general enough to be reused for other microprocessors, particularly another member of the AAMP family. This approach was used to formally specify the entire microarchitecture, more than half of the instruction set, and to verify a core set of eleven AAMP5 instructions representative of several instruction classes. The methodology and the formal machinery developed are adequate to cover all of the remaining AAMP5 instructions. Although PVS was the vehicle of the experiment, the approach could also be used with other sufficiently powerful theorem provers.

Another key result of the project was the discovery of both actual and seeded errors. Two actual microcode errors were discovered during development of the formal specification, illustrating the value of simply creating a precise specification. Both were specific to the AAMP5 and were corrected before first fabrication. Two additional errors seeded by Collins in the microcode were systematically uncovered by SRI, who knew that bugs had been seeded but not their location or identity, while doing the correctness proofs. One of these was an actual error that had been discovered by Collins after first fabrication but left in the microcode provided to SRI. The other error was designed to be unlikely to be detected by walk-throughs, testing, or simulation.

Several other technical issues emerged during the project that influenced the transfer of formal verification to practice. The first was the importance of choosing an appropriate problem for technology transfer. Microprocessor designs are a good candidate since their verification has been studied for a number of years and their requirements are usually well defined. Another important issue is the support available from the proof-checker to efficiently automate the construction of proofs. The PVS proof-checker is designed to make very heavy use of symbolic rewriting and a number of decision procedures, effectively automating most of the low-level inferences encountered during mechanical proofs. Even so, the PVS rewriting engine had to be optimized in a number of ways for the purpose of this project. Other issues included the need to select a style of specification that would support both efficient proving and review by the engineers, the need for libraries of general-purpose theories, and the need to engineer proof strategies for reuse.

This paper discusses both the technical and technology transfer aspects of the project. Section 2 provides background information and discusses the scale of the project. Section 3 describes the AAMP5 microprocessor at both the instruction set and register transfer levels.

Section 4 describes the project organization, metrics, and the working model used to ensure effective technology transfer from SRI to Collins. Sections 5 and 6 describe how the macro and micro architectures were specified. Section 7 describes the formalization of the correctness of the AAMP5 pipeline and the approach used to mechanize the proofs of correctness. Conclusions are presented in the last section.

2. Background

NASA Langley's research program in formal methods [6] was established to bring formal methods technology to a sufficiently mature level for use by the United States aerospace industry. Besides the inhouse development of a formally verified reliable computing platform RCP [10], NASA has sponsored a variety of demonstration projects to apply formal methods to critical subsystems of real aerospace computer systems.

The Computer Science Laboratory of SRI International has been involved in the development and application of formal methods for more than twenty years. The formal verification systems EHDM and the more advanced PVS were both developed at SRI. Both EHDM and PVS have been used to perform several verifications of significant difficulty, most notably in the field of fault-tolerant architectures and hardware designs. Recently, SRI has been actively involved in investigating ways to transfer formal verification technology to industry.

Collins Commercial Avionics is a division of Rockwell International and one of the largest suppliers of communications and avionics systems for commercial transport and general aviation aircraft. Collins' interest in formal methods dates from 1991 when it participated in the MCC Formal Methods Transition Study [12]. As a result of this study, Collins initiated several small pilot projects to explore the use of formal methods, with verification of the AAMP5 microcode being the latest and most ambitious in the series.

2.1. AAMP family of microprocessors

The Advanced Architecture Microprocessor (AAMP) consists of a Rockwell proprietary family of microprocessors based on the Collins Adaptive Processor System (CAPS) originally developed in 1972 [1, 23]. The AAMP architecture is specifically designed for use with block-structured, high-level languages such as Ada in real-time embedded applications. It is based on a stack architecture and provides hardware support for many features normally provided by the compiler run-time environment, such as procedure state saving, parameter passage, return linkage, and reentrancy. The AAMP also simplifies the real-time executive by implementing in hardware such functions as interrupt handling, task state saving, and context switching. Use of internal registers holding the top few elements of the stack provides the AAMP family with performance that rivals or exceeds that of most commercially available 16-bit microprocessors.

The original CAPS architecture, a multiboard minicomputer, was developed in 1972 and was quickly followed by the CAPS-2 through CAPS-10. In 1981, the original AAMP consolidated all CAPS functions except memory on a single integrated circuit. It was followed by the AAMP2, AAMP3, and AAMP5. Members of the CAPS/AAMP family have been used in an impressive variety of products including the Lockheed L-1011 Digital Flight Control System (DFCS) and Active Control System (ACS), the Boeing 757 and 767 Autopilot Flight Director System (AFDS), the Boeing 747-400 Integrated Display System (IDS) and Central Maintenance Computer (CMC), the Boeing 757, 767, and 737-300 Electronic Flight Instrumentation System (EFIS) and Engine Instrumentation/Crew Alerting System (EICAS), and Navstar Global Positioning System (GPS) receivers.

The AAMP5 was designed as an object-code-compatible replacement for the earlier AAMP2 [23], with advanced implementation techniques such as pipelining providing a more than threefold performance improvement. The AAMP5 is designed for use in critical applications such as avionics displays, but is not intended for use in ultra-critical systems such as autoland or fly-by-wire.

2.2. *PVS*

PVS (Prototype Verification System) [26] is an environment for specification and verification that has been developed at SRI International's Computer Science Laboratory. In comparison to other widely used verification systems, such as HOL [15] and the Boyer-Moore prover [2], the distinguishing characteristic of PVS is that it combines a highly expressive specification language with a very effective interactive theorem prover in which most of the low-level proof steps are automated. The system consists of a specification language, a parser, a typechecker, and an interactive proof checker. The PVS specification language is based on higher-order logic with a richly expressive type system so that a number of semantic errors in specification can be caught by the typechecker. The PVS prover consists of a powerful collection of inference steps that can be used to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps involve, among other things, the use of arithmetic and equality decision procedures, automatic rewriting, and BDD-based boolean simplification. PVS provides a simple language to compose the primitive proof steps into more complex proof strategies.

3. The AAMP5 microprocessor: An informal description

This section provides an informal discussion of the AAMP5 at both the instruction set (macroarchitecture) and register transfer (microarchitecture) levels. Since the members of the AAMP family share similar instruction sets, we will refer to the AAMP macroarchitecture and the AAMP5 microarchitecture.

3.1. *The macroarchitecture*

At the instruction set level, the main features that make the specification and verification of the AAMP challenging are its support for multi-tasking and error handling, the process stack, and the variety of its CISC-like instructions. We briefly discuss these features below. A more detailed discussion can be found in [1].

The AAMP provides separate address spaces for *code memory* and *data memory*. While not required by the AAMP architecture, code memory is typically implemented in ROM. Both code and data memory are segmented, with code memory organized into 512 *code environments*, each containing 64 K 8-bit bytes of code. Data memory is organized into 256 *data environments*, each containing 64 K 16-bit words of data.

The process stack is central to the AAMP macroarchitecture, implementing in hardware many of the features needed to support high-level block structured languages and multitasking. Each task maintains a single process stack in the task's data environment. Many of the internal registers maintained by the AAMP are used to define the process stack. The DENV (data environment) register contains the pointer to the data environment of the current active task. The TOS (top of stack) register points to the topmost word in the process stack and the SKLM (stack limit) register points to the lower limit beyond which the stack is not allowed to grow (the stack grows downward in memory). The LENV (local environment) register points to the local environment of the current procedure and is used in addressing local variables. The CENV (code environment) and PC (program counter) registers point to the code environment and address of the next instruction to be executed.

The AAMP instruction set (209 instructions) is large and CISC-like. It closely resembles the intermediate output of many compilers, directly supporting high-level language constructs such as procedure calls and returns. Instructions vary in length from 8 to 56 bits, although most are only 8 bits long, yielding improved throughput and code density. The instruction set supports a variety of data types, including 16- and 32-bit integer, 16- and 32-bit fractional, 32- and 48-bit floating-point, and 16-bit logical variables. The floating-point formats, which were not considered in the verification, have an 8-bit, excess-128 exponent and, respectively, 24 and 40 bits of fractional mantissa. Addition, subtraction, multiplication, division, and type conversions are provided for all of the arithmetic types. Computational exceptions, such as arithmetic overflow and divide-by-zero, are detected and handled automatically.

3.2. The microarchitecture

The internal microarchitecture of the AAMP5 employs four major, semiautonomous, functional units as shown in the block diagram of figure 1.

The Data Processing Unit (DPU), where the microcode resides, provides the data manipulation and processing functions to execute the AAMP5 instruction set. Instruction fetching is performed by the Lookahead Fetch Unit (LFU) by maintaining a 4-byte-long instruction queue that it endeavors to keep full. In addition to prefetching into the instruction queue, the LFU parses the instruction stream, assembles immediate operands (IM), and provides the DPU with a microprogram entry point (EP) translated from the instruction opcode. The LFU also passes the program counter (DP) associated with the parsed instruction. The AAMP5 includes 1024 bytes of direct-mapped instruction cache. Because the operational details of ICU are hidden from the DPU, the LFU and the ICU are shown in figure 1 enclosed inside a shell with which the DPU interacts directly. The Bus Interface Unit (BIU) performs external fetchs for the LFU, as well as data reads and writes initiated by the DPU.

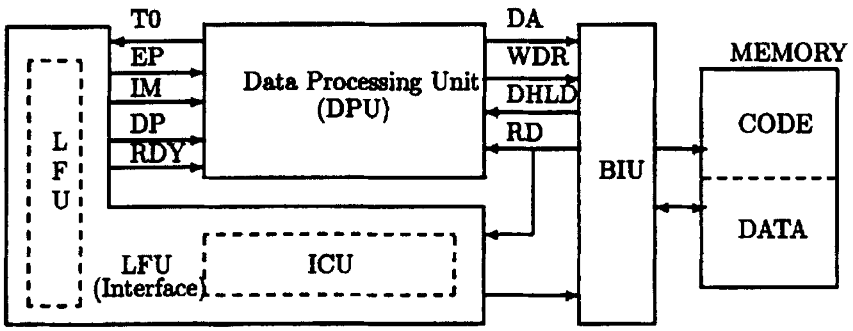


Figure 1. AAMP5 block diagram.

The DPU provides the data manipulation and processing functions required to execute the AAMP5 instruction set. A block diagram of the DPU is shown in figure 2, which also includes the details of the LFU. The DPU has two main parts: the datapath and the microcontroller.

*3.2.1. **The datapath.*** The datapath consists of the components that perform the required data manipulation for instruction execution. It includes a 10-word multi-port register file, a 48-bit ALU, shifters, multiplication support, and several registers. Some of the registers (R and W) are used to hold the operands of the ALU and some are used to hold the results (T, Q, and M triplets) after ALU and shift operations.

The ALU and the Reg File (register file) are two of the most complex units in the DPU. The ALU supports a large number of logical and 2's complement and 1's complement arithmetic operations on 48-bit bit-vectors. Besides the two bit-vector operands, the ALU accepts a carry-in input and produces a carry-out. The ALU can, if necessary, bypass the operand registers (R and W) and get its operands directly from the result registers (T, Q, and M).

The Reg File is a complex multi-port register file that can be read (and written) simultaneously from (into) up to three registers relative to a read (write) index controlled by the microcode. It contains bypass logic to use the write value instead of the contents of a register during a read if there is a read-write clash. Six designated registers (STK0 through STK5) in the register file (along with the T registers) are used to implement the stack cache as described in Section 3.2.3. The remaining registers in the register file are used as pointers to the code and data memory such as the top-of-stack and data environment pointer (DENV).

*3.2.2. **The microcontroller.*** The microcontroller generates the signals that control the movement of data within the datapath as well as the next microinstruction to be read from the ROM. It contains the microprogram store, the microsequencer, which computes the next microaddress, and the microinstruction control registers (MC0, MC1, and MC2). AAMP5 instructions are executed in a 3-stage—*fetch*, *setup*, *compute*—pipeline (figure 2), where the fetch stage consists of the activities that take place prior to an instruction moving into the

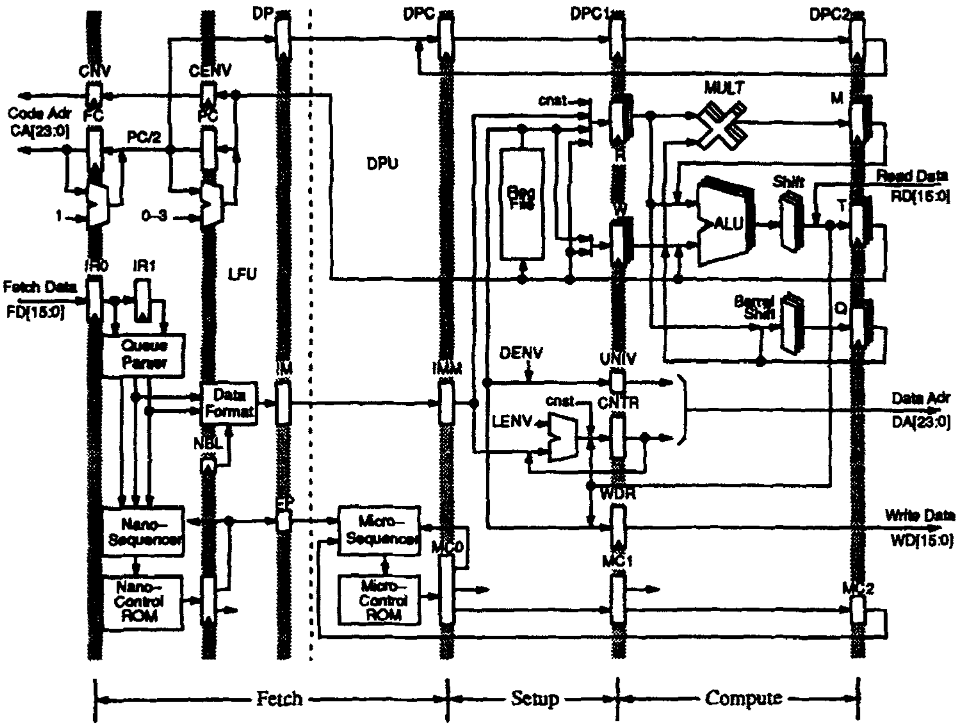


Figure 2. The LFU and DPU datapath.

DPU from the LFU. The pipeline and its stages are described in more detail in Section 7.2. The activities performed in the pipeline stages are controlled by a microinstruction as it moves from MC0 to MC2. The program counter DP is also pipelined using a sequence of registers so that DPC, DPC1, and DPC2 contain the program counters corresponding to the instructions in MC0, MC1, and MC2, respectively.

3.2.3. The stack cache and stack adjustment. Conceptually, the process stack consists of an area of data memory, a stack limit (SKLM) register, and a *logical* top-of-stack (TOS) register. In actuality, the DPU holds up to nine words of the top of the process stack in a set of internal registers called the *stack cache*, with the rest being stored in memory. The *T* register-triplet always holds the latest data item, which can be one, two, or three words long, pushed on to the top of the stack. The rest of the stack cache elements are stored in the register file. The DPU maintains the *physical* top-of-stack (TOS) register marking the boundary between the portion of the stack implemented in data memory and the portion implemented in the cache and the stack cache occupancy information. Note that the logical top-of-stack is the sum of physical TOS and the stack cache occupancy.

The occupancy state of the stack cache that would result if an instruction were completed is maintained in a pair of registers for each of the AAMP5 instructions being executed in

MC0 (SV and TV) and MC1 (SV1 and TV1). (The SV-TV register ensemble and the logic that maintains stack occupancy are not shown in figure 2.) TV contains the number of words that would be occupied in the *T* register triplet and SV contains the number of registers in the register file that are used for the stack cache. When an instruction to be executed requires more operands than are available in the cache, the processor automatically reads additional locations from memory to the stack cache. Similarly, when an instruction will place more operands on to the stack than there are empty spaces in the stack cache, the processor automatically writes the cache register contents to memory.

The process of maintaining the proper number of operands in the stack cache is called a *stack adjustment*. Prior to loading into MC0 the first microinstruction of an instruction, the DPU compares the stack cache occupancy state corresponding to MC0 against the *entry condition code* of the instruction. This code, which is stored for every instruction in a read-only-memory, gives the stack cache requirements for the instruction. If the stack cache occupancy is not acceptable, the DPU automatically executes microcoded push or pop stack adjustments, causing a minimum number of 16-bit words to be written to or read from the active process stack area in memory to satisfy the entry condition of the instruction.

4. Project organization

Formal verification of the AAMP5 microcode was selected for this project for a number of reasons. Both Collins and SRI wanted to explore the usefulness of formal verification on an example that was large enough to provide realistic insight, yet small enough to be completed at reasonable cost. Verification of the AAMP5 microcode fit these criteria well. While the AAMP5 was one of the most complex microprocessors Collins had built, its requirements were well understood since it was to be object-code-compatible with the earlier AAMP2. This allowed the formal methods team to concentrate on formal specification and verification rather than on designing a new product. Also, much of the complexity of an AAMP microprocessor resides in the microcode, and past experience has shown that this is one of the most difficult parts of the microprocessor to get right. Success with formal verification in other significant projects suggested that this technology might be ready for application to an industrial microprocessor.

Because of the importance of the AAMP5 to Collins' product line, the formal specification and verification of the AAMP5 were performed as a shadow project and did not replace any of the normal design and verification activities performed on a new microprocessor. This parallel approach also allowed us to relax some of the steps that would be required on a production project and focus instead on the application of formal methods. To fit the scope of the project to the time available, a core set of 13 instructions, each representative of a class of AAMP instructions, was identified to be specified and verified by SRI. An additional set of 11 instructions was identified to be specified and verified by Collins as time permitted. Even so, it was necessary to specify the entire AAMP5 architecture and develop the infrastructure needed to verify the entire instruction set since the core set contained at least one member from each of the major instruction classes.

A summary of the level of effort is presented in Table 1. As shown, relatively little time was spent on training the Collins' engineers in PVS. The small amount of structured training

Table 1. Level of effort.

Task	Performed	Start	Stop	Hours
Project management				
Planning & monitoring	Collins	Jan 93	Aug 94	123
Education				
PVS course	Collins	Feb 93	Feb 93	125
	SRI	Feb 93	Feb 93	68
Specification of the macroarchitecture (2,550 lines of PVS)				
Initial development	Collins	Mar 93	May 93	172
	SRI	Mar 93	May 93	360
Revision & extension	Collins	May 93	Sept 93	289
	SRI	May 93	Sept 93	120
Inspection	Collins	Sept 93	Feb 94	96
Resolve inspection issues	Collins	Feb 94	May 94	64
Revision to support proofs	Collins	Mar 94	Aug 94	54
Specification of the microarchitecture (2,679 lines of PVS)				
Initial development	Collins	May 93	Feb 94	137
	SRI	May 93	Feb 94	520
Revision	Collins	Feb 94	Aug 94	160
	SRI	Feb 94	Aug 94	120
Inspection	Collins	Mar 94	Aug 94	83
Resolve inspection issues	Collins	Mar 94	Aug 94	66
Translate microcode to PVS	Collins	Jun 94	Aug 94	21
Revision to support proofs	Collins	Jun 94	Aug 94	12
Proofs of correctness				
Development of correctness criteria	SRI	Mar 94	Jun 94	320
Developing proof infrastructure	SRI	May 94	Aug 94	240
Verification of core instructions	SRI	Jun 94	Aug 94	240

needed was one of the surprises of the project. Early on, SRI conducted a one-week course on the use of PVS and formal specifications at the Collins Cedar Rapids facility for the five engineers who would be involved with the project. The course consisted of five half-day lectures with related lab exercises in the afternoon. No additional formal training seemed necessary. When new team members joined the project, they were provided access to the PVS documentation and trained by inclusion in review of the PVS specifications. The most effective form of education seemed to be hands-on development with frequent peer review.

Aside from overall management and education, the project split naturally into three phases: specification of the macroarchitecture (Section 5), specification of the microarchitecture (Section 6), and proofs of correctness of the microcode (Section 7). The basic process followed in the first two phases was that Collins would provide design specifications to SRI, SRI would provide first drafts of PVS specifications to Collins, and Collins would informally review these specifications and return comments to SRI for revision. At some point, the Collins team would take the specifications, prepare them for formal inspections [11], conduct the inspections, correct the defects found, and send the revised specifications back to SRI. This approach was chosen both to validate the correctness of the specifications and to ensure that Collins personnel became actively involved in developing the PVS specifications. A similar process was followed for performing proofs of correctness of the microcode.

To reduce the potential for missing errors in the microcode due to errors in the PVS specifications, independent teams were assigned to different portions of the project. While all early drafts of the specifications were produced by SRI, different individuals at Collins were assigned to review and revise the macroarchitecture and microarchitecture specifications. Different teams were also used to inspect the macroarchitecture and the microarchitecture. The microcode itself was produced by a member of the original AAMP5 team without any knowledge of the formal specifications and translated into PVS by yet another individual. As a result, the process of proving the microcode correct often revealed errors in the specifications, but once a proof was completed, confidence in the correctness of the associated microcode was high.

4.1. Development of the macroarchitecture specification

The initial specification of the AAMP macroarchitecture was developed through gradual refinement by SRI and Collins and resulted in several iterations, each incorporating increasing amounts of detail. Since the AAMP5 was to be object-code compatible with the earlier AAMP2, this work was based on the AAMP2 Reference Manual [23]. Each iteration was reviewed via informal walkthroughs by Collins and the comments returned to SRI. This phase lasted approximately three months, took 532 man hours to complete, and resulted in a first draft of the specification consisting of 1,595 lines of PVS organized into 25 theories[1].

Once SRI and Collins were satisfied with the overall structure of the specification, its completion was taken over by Collins. This was done for several reasons, the most pragmatic being to allow SRI to move on to the specification of the micro-architecture. Ownership by Collins also encouraged transfer of the formal methods technology. There was also growing concern whether the AAMP domain experts, who were not skilled in PVS, would be able and willing to read the PVS specification. It was felt the Collins team was best situated to facilitate this.

Over the next five months, the roles of SRI and Collins on the macroarchitecture were reversed, with Collins revising and extending the specifications and SRI providing informal review. More of the executive service functions were specified and the number of instructions specified was increased to 108 of the AAMP's 209 instructions. NASA Langley also took over completion and validation of the bit vector theories. To make the specifications more accessible to the AAMP domain experts, considerable effort was invested in improving their readability by choosing more meaningful names, adding general comments, adding comments tracing the specifications back to the AAMP2 Reference Manual, and ensuring that all functions were written as clearly as possible. Approximately 409 man hours were invested in this effort. At its conclusion, the macroarchitecture specification consisted of 2,550 lines of PVS organized into 48 theories, not including the bit vectors library discussed earlier.

Formal inspection [11] of the macroarchitecture was felt to be essential, both to validate the correctness of the specification and to familiarize more engineers at Collins with PVS. In preparation for these inspections, an overview of the specifications was presented in four reviews of two hours each. At the end of these sessions, the engineers were divided into two teams, one that would review the macro-level specifications and one that would review the micro-level specifications. Checklists were drawn up for use in the inspections based on earlier checklists used in inspecting VDM [17] specifications, the RAISE Method Manual [3], and checklists used for code inspections at Collins.

Eleven inspections were held of the macro level specification covering thirty-one of the most important theories. Inspectors were required to review the designated theories ahead of time, using the checklists as guides, and record all potential defects encountered. Defects were classified as trivial, minor, and major. Trivial defects were defined as those that did not affect correctness and for which an obvious solution existed, such as spelling errors. Minor defects included those that might affect clarity or maintenance but did not affect correctness. As a rule of thumb, a defect was classified as minor if two reasonable people could disagree on whether it was a defect. Major defects were defined as those that affected correctness and obviously should be corrected. Despite their name, most of the major defects were very limited in scope and could be corrected in a few minutes. Some of the errors were misunderstandings by SRI, some were errors in the original AAMP documentation, and some had been inserted by Collins during the revisions. During the inspections, each inspector presented the minor and major defects they had found. While consensus of the team was required for a defect to be officially recorded, the majority of issues raised were recorded as defects. Later, each defect recorded was corrected. Total time spent in preparation by all participants, time spent in inspection, and number of defects found are shown in Table 2.

During the first inspection, team members were still uncomfortable with PVS, as indicated by the number of hours spent in preparation. This apprehension dissipated quickly and the

Table 2. Macroarchitecture inspection results.

Inspection #	# PVS theories	Lines of PVS	Inspectors	Preparation (hours)	Inspection (hours)	Minor defects	Major defects
1	3	203	3	12.0	0.8	6	1
2	3	281	3	3.0	0.8	6	6
3	4	216	3	4.0	0.8	12	3
4	2	116	3	4.3	1.0	3	5
5	2	195	3	3.5	1.0	5	6
6	3	149	3	3.0	0.5	3	2
7	4	147	3	2.7	0.4	6	2
8	3	135	3	3.5	0.6	4	3
9	3	332	3	1.5	0.5	5	0
10	2	204	3	1.2	0.4	1	0
11	2	079	3	0.9	0.3	2	0
Total	31	2057		39.6	6.3	53	28

inspectors settled down to a rate of approximately 150 lines of PVS and comments per hour of preparation time (the inspection rate increased during inspections nine through eleven since these were of simple tables). This rate is probably somewhat high since the inspectors were well aware that this was a shadow project. On an actual project, more preparation time would have been required. Even so, 53 minor (style) defects and 28 major (substantive) defects were discovered in specifications that had been carefully prepared prior to inspection. As shown in Table 1, approximately 96 hours were spent conducting the inspections and 64 hours were spent correcting the defects found.

4.2. *Development of the microarchitecture specification*

Development of the microarchitecture specification mirrored that of the macroarchitecture. As indicated in Table 1, initial development of the microarchitecture specification by SRI took approximately 657 man hours over 10 months. The specification closely followed the block structure of the microarchitecture, usually with one theory per component. Surprisingly, the specification of the microarchitecture without the PVS version of the microcode was only slightly larger than the specification of the macroarchitecture, an indication how much of the complexity of the AAMP5 is contained within the microcode.

To prepare the PVS specifications for inspection, the initial microarchitecture specifications developed by SRI were informally reviewed by three Collins engineers familiar with both PVS and the AAMP5 and revised as was done for the macroarchitecture. Since the initial specifications covered the entire microarchitecture, there was no need for Collins to extend the specification. Most of the changes consisted of modifying names to reflect local conventions, adding comments tracing back to the design documents, and improving the clarity of the specifications. Even so, this was a sizeable effort, requiring 280 hours spread over seven months. When completed, the microarchitecture specification consisted of 2,679 lines of PVS and comments organized into 20 theories.

Revisions were also made later to the microarchitecture specifications to facilitate proofs, but these were more technical in nature and not as significant as the ones made to the macroarchitecture specification. Most involved trading the use of an advanced and expressive construct of the specification language for a more basic construct to improve the efficiency of proofs.

Formal inspections of the microarchitecture were conducted just as for the macroarchitecture. To maximize the independence between the macro and micro specifications, only one participant from the macroarchitecture inspections was included in the microarchitecture inspection team. Ten inspections were held, including two reinspections, covering 15 of the most important theories. The results are shown in Table 3.

Again, the inspectors quickly adapted to PVS, reaching an average inspection rate of approximately 290 lines of PVS and comments per hour. Interestingly enough, the designers of the AAMP5, who were the least familiar with the PVS language, found the specifications the simplest to read, consistently turning in the most major defects and the lowest preparation times. This was a direct result of their detailed knowledge of the AAMP5 and the close correspondence between the AAMP5 design and the specifications. As with the macroarchitecture specification, more preparation time would have been required on an

Table 3. Microarchitecture inspection results.

Inspection #	# PVS theories	Lines of PVS	Inspectors	Preparation (hours)	Inspection (hours)	Minor defects	Major defects
1	3	357	5	8.3	1.5	15	6
2	2	173	5	3.9	1.0	11	2
3	1	146	4	3.2	0.4	7	1
4	1	146	5	3.3	0.8	6	2
5	1	152	5	4.2	0.3	3	4
6*	1	160	4	1.3	0.5	1	0
7	1	272	4	3.6	0.6	10	1
8*	1	423	5	4.6	0.8	4	2
9	3	197	4	2.0	0.5	6	1
10	3	256	4	1.6	0.5	1	0
Total	17	2282		36.0	6.9	64	19

*Reinspection of previously inspected theory.

actual project. Sixty-four minor (style) defects and 19 major (substantive) defects were discovered. As shown in Table 1, 83 hours were invested in conducting the inspections and 66 hours were spent correcting the defects found.

4.3. Development of the correctness proofs

The correctness proofs were developed in much the same way as the micro and macroarchitecture specifications. SRI first formalized the basic notion of correctness for the AAMP5 as described in Section 7 and explored how the proofs should be organized by proving the ADD instruction correct. Due to the complexity of the AAMP5, this was a significant effort, requiring approximately 320 hours of effort.

It became clear during this phase that additional organization of the proof structure was necessary if the techniques were going to be successfully transferred to Collins. Consequently, SRI invested considerable effort into decomposing the proof process into two steps (Section 7). The first step establishes the correctness of the microcode at the microarchitecture level, breaking it down into general verification conditions that are the same for every macro instruction and instruction specific verification conditions that define how the microarchitecture state changes for a particular instruction. The second step relates these proofs to the macroarchitecture specification via the abstraction function. These steps were then captured as PVS strategies and packaged for reuse. This phase required approximately 240 hours of effort.

Finally, the proofs for 11 instructions were completed by SRI using the strategy outlined above. This phase required an additional 240 hours. As part of the technology transfer, a Collins engineer helped set up the verification conditions for one instruction and completed the proof of some of the verification conditions. Also, another engineer at Collins used the

PVS prover to develop the descriptive macro lemmas from the constructive specification of the macroarchitecture (Section 5).

4.4. *Observations*

Several trends became apparent during the development of the macro and microarchitecture specifications. First, as the specification grew in size, an ever increasing portion of it was devoted to defining the properties of bit vectors, i.e., sequences of bits such as words of memory and internal registers. Ultimately, these theories evolved into a reusable library of 2,030 lines of PVS organized into 31 theories. Availability of this library at the start of the project would have greatly shortened this phase.

Also, large parts of the specification were simply tables of attributes of the various AAMP instructions. While the PVS representation of this information was readable, a PVS construct explicitly designed to support the expression of tabular data would have improved their clarity. Such a construct has been added to the latest version of PVS.

Completion of the microarchitecture specification took longer than the specification of the macroarchitecture for a number of reasons. The AAMP5 microarchitecture document, unlike the AAMP2 Reference Manual, is targeted for an audience that is familiar with the basic architecture of the AAMP processor family. As a result, SRI had to spend considerable time becoming familiar with the design. In particular, the interactions between the DPU and its environment were difficult to specify. Although the design of the LFU and the BIU were well documented, the interface conditions that the DPU has to obey to ensure proper LFU and BIU services were not explicitly stated. This information had to be extracted through reverse engineering of the detailed designs and extensive discussions with the Collins staff. The time spent on both these efforts could have been substantially reduced if formal specification activity were integrated earlier and more closely into the conventional design cycle.

Perhaps the greatest surprize was the ease with which the inspectors became comfortable with PVS. This was particularly notable since very little formal training was provided. To a large extent, this was due to the gradual, stylistic, and example-oriented fashion in which the language was introduced and the time spent preparing the theories for inspection. PVS uses a syntax that is close to conventional set-theoretic notation and supports a type system that is similar to the one found in a modern high-level programming langauge. These features of PVS certainly helped in its acceptance since the enginers at Collins were sensitive about the presentation of the specification. The specification uses higher-order functions and some of the advanced features, such as dependent types, quite extensively. Every such unconventional feature was introduced only after carefully motivating its usage via examples. Once the fetaures were well-illustrated, the engineers did not have much difficulty in using them.

While the purpose of the inspections was to validate the accuracy of the formal specifications, issues of style and clarify could dominate an inspection if a theory was not well organized. On the few occasions that unprepared theories were submitted to inspection, the result was quick rejection by the inspection team. Clear organization, standard naming conventions, and meaningful comments were essential.

Also surprising was the extent to which formal specifications and inspections complemented each other. The inspections were improved by the use of a formal notation, greatly reducing the amount of debate over whether an issue really was a defect or a personal preference. In turn, the inspections served as a useful vehicle for education and arriving at consensus on the most effective styles of specification. This is reflected in Tables 2 and 3 by the lower number of defects recorded in the later inspections.

It was more difficult to transfer the verification technology to Collins than the formal specification technology. In large part, this was because the formal specifications were written early in the project while the approach for formal verification was developed towards the end when time and funds were running low leaving insufficient time to package the proof process as effectively as was done with the specification techniques for a nonexpert. In addition, the size of the combined micro and macroarchitecture specifications strained the automatic rewriting capabilities of the original version of PVS. Enhancements to PVS, especially optimizations to its rewriting engine, that were incorporated as a result of our experiments with the proofs of the first few instructions helped to relieve these problems. One of the engineers on the Collins team visited SRI for a week to study and learn about the proof process. During the visit, he was able to formulate and partially verify an instruction after studying a completed proof of a related instruction. Our idea is to package the proof process so that an engineer will be using the prover as anintelligent symbolic simulator.

The AAMP5 experiment was typical of many large, industrial projects in that the same fundamental constructs appear over and over. While a substantial amount of groundwork was necessary to automate the verification of the microcode, once the basic framework was established it was possible to bring the costs down to a reasonable level and for practicing engineers to participate in the proof process. In our case the groundwork involved not only decomposing the verification problem into mangeable units, but also developing the formal infrastructure needed to define and automatically reason with bit-vectors. We believe most of this framework can be reused in the verification of another microprocessor. One of the primary goals of the follow-on project underway is to ensure the transfer of the formal verification technology and determine how much of the AAMP5 groundwork can be reused.

5. Formal specification of the AAMP5 macroarchitecture

Our approach to specifying the AAMP5 macroarchitecture is similar to the one used in most earlier microprocessor verification. The state of the macromachine is modeled as a record that includes external memory and the internal state that affects its observable behavior, such as the internal registers defining the process stack and the program counter. The AAMP5 instructions are specified by defining a next-state function that characterizes the effect of executing the "current" instruction pointed to by the program counter. The behavior of all instructions in the absence of external interrupts can be specified in this fashion.[2]

The PVS specification of the macro state is shown in figure 3. Code memory and data memory are defined separately, since this is the conceptual model presented to a programmer. While it is possible for an external memory unit to implement code and data memory in the same address space, and even to overlap code and data memory, code memory is normally

```
% The macro state theory defines the state of AAMP internal registers, code memory, and data memory.
% Basic operations based on the combined state of these registers and memory are also defined here.

macro_state: THEORY

BEGIN
  IMPORTING opcodes_attributes, code_memory, data_memory

  % The macro state of the AAMP consists of code memory, data memory,the user/executive mode indicator,
  % the interrupt enabled indicator,the cenv, pc, denv, sklm, lenv, and tos registers.

  macro_state: TYPE = [# cmem  : code_memory,
                         dmem  : data_memory,
                         user  : bool,
                         inte  : bool,
                         cenv  : word,
                         pc    : word,
                         denv  : word,
                         sklm  : word,
                         lenv  : word,
                         tos   : word #]

  % Pushes a word on the process stack. If full, the tos is decremented but the word is not written.
  push(wd:word, st:macro_state):macro_state =
    IF tos(st) > sklm(st)   THEN
       st WITH [(dmem)(word2denv(denv(st)))(tos(st)-1) := wd,
                (tos)  := tos(st) - 1 ]
     ELSE
       st WITH [(tos)  := tos(st) - 1 ]
     ENDIF
                             ...
END macro_state
```

Figure 3. The AAMP5 macrostate.

implemented in ROM and is physically distinct from data memory. The remaining items in the macrostate define the CENV, PC, DENV, SKLM, LENV registers and the USER and INTE flags discussed in Section 3.1. The macrostate theory also defines a number of auxiliary functions that manipulate the macrostate to make later theories more readable. For example, `push` pushes a word on the current process stack.

The next state function (`next_macro_state`) takes an arbitrary macrostate and returns the macrostate that would result after the current instruction is executed. The next state function is defined for each instruction class in an `update` theory for that class. For example, the next state function for for the arithmetic instructions is defined in the `arith_update` theory, a portion of which is shown in figure 4. Arithmetic instructions perform a specific operation, such as addition or subtraction, on the top few elements of the accumulator stack and push the results onto the stack after consuming the operands. Two functions are used in the definition of the `next_macro_state` function for the arithmetic operations. The `exception_number` function determines if the instruction will result in an exception given the current macrostate. The `normal_result` function defines the change in state returned by the `next_macro_state` function when an exception does not occur. If an exception does occur, this state and the exception number are given as parameters to the `exception_macro_state` function, defined in the `exceptions` theory, that pushes the exception number on top of the (erroneous) result and invokes the exception handler.

```
% The arith_update theory defines the change on the macro state caused by arithmetic instructions
% in absence of stack overflow. The instruction result is computed and pushed on the stack.
% If the instruction results in an exception, the state defined by the exceptions theory is returned.

arith_update: THEORY
  BEGIN
    IMPORTING exceptions

    % Subtype of macro state in which current instruction is an arithmetic instruction.
    arith_state: TYPE = {st: macro_state|instruction_class_of(current_opcode(st)) = arithmetic}

    % Returns the exception number associated with an instruction.
    % Zero is used to indicate the absence of an exception.
    exception_number(st:arith_state): below[29] =
      LET nstkwds = number_of_stack_words(current_opcode(st)),
          args    = top_elements(st,nstkwds)
       IN IF current_opcode(st) = ADD THEN
            IF overflow(args(0), args(1)) THEN 7 ELSE 0 ENDIF
          ELSIF current_opcode(st) = ADDD THEN
            IF overflow(args(1) o args(0), args(3) o args(2)) THEN 8 ELSE 0 ENDIF
          ELSE 0 ENDIF

    % Returns the normal result obtained in the absence of an exception.The arguments to the instruction
    % are popped from the stack, the results are pushed on the stack and the program counter advanced to
    % the next instruction.
    normal_result(st: arith_state): macro_state =
      LET nstkwds = number_of_stack_words(current_opcode(st)),
          args    = top_elements(st, nstkwds),
          popped  = multipop(st, nstkwds)
      IN IF current_opcode(st) = ADD THEN
            push(args(0) + args(1), popped)
          ELSIF current_opcode(st) = ADDD THEN
            LET result = (args(1) o args(0)) + (args(3) o args(2))
            IN push(result^(31, 16), push(result^(15, 0), popped))
          ELSE st  ENDIF WITH [(pc) := next_pc(st)]

    % Returns the next macro state for an arithmetic instruction.
    next_macro_state(st: arith_state): macro_state =
      IF exception_number(st) = 0 THEN normal_result(st) ELSE
         exception_macro_state(normal_result(st),exception_number(st))
      ENDIF

  END arith_update
```

Figure 4. PVS specification of arithmetic instructions.

Additional details on how the AAMP5 interrupts can be specified and verified can be found in [28]. Two issues, one general and the other specific to AAMP5, that arose in the development of macroarchitecture specification are discussed in the next two sections.

5.1. *Constructive vs. descriptive specifications*

One of the early choices to be made in specifying the macro-architecture was whether to specify the `next_macro_state` function *constructively*, i.e., by explicitly defining how the result of the function is to be constructed, or *descriptively*, i.e., by stating a set of properties (axioms) that the function is to satisfy. The main advantage of a constructive style of specification is that the PVS language mechanisms will ensure that the function is not only well-defined but total. A disadvantage is that it is difficult to leave parts of the specification deliberately underspecified. A descriptive style is naturally suited for underspecification, but makes it possible to introduce inconsistent axioms. In addition, descriptive specifications are often more difficult to understand than constructive specifications for an uninitiated reader. Most earlier processor verification efforts have used a constructive style of specification.

We initially chose the constructive style since (besides the reasons given above) many of the AAMP instructions are similar enough (for example, a REFS and a REFSL differ only in the addressing mode) that they could be compactly specified using a constructive style. Also, much of the most complex behavior of the AAMP instructions was already specified at Collins using a procedural style (in the form of pseudo-code) that could be easily translated into a constructive specification. One result of this choice was to make the effort required to specify a single instruction quite high since most of the infrastructure for an entire class of instructions was required to complete the first instruction in that class. A benevolent side-effect was that this made it easy to specify far more instructions (108) than the 13 instructions originally planned.

However, while doing the correctness proofs it became clear that a more descriptive style of specification in which the change in state was defined more directly would be helpful as an intermediate step in the proof. Fortunately, these could be stated as lemmas that could be proven from the original specification, preserving our investment in the original specification. As these lemmas were created, it became evident that they were in many ways a preferable style of specification. They were more readable, simpler to validate, and were closer to what a user wanted to know in the first place. They also made it possible to specify a small portion of the `next_macro_state function`, i.e., to specify one instruction or part of an instruction at a time. An example of the descriptive style of specification is presented below for the ADD instruction for the case when the addition does not overflow. The other cases are characterized similarly by other lemmas. In the constructive style, all the cases were embedded inside a function definition in an algorithmic style as shown in figure 4. We believe that the advantages of the descriptive style outweigh its disadvantages enough to use it as the preferred style. The likelihood of inconsistency in the specification can be eliminated by using simple syntactic guidelines that keep the axioms from semantically overlapping each other.

```
% ADD - Two's Complement Add, Single Word (no exception).The two words on the top of the
% stack are added, with the resulting sum replacing them on the top of the stack.

ADD_lemma_1: LEMMA
  LET X = current_data_env(st)(tos(st)+1),
      Y = current_data_env(st)(tos(st))
  IN current_opcode(st) = ADD & arith_update.exception_number(st)=0  =>
     normal_macro_machine.next_macro_state(st) =
       st WITH [(dmem)(word2denv(denv(st)))(tos(st)+1) := X + Y,
               (pc)  := pc(st)  + 1,
               (tos) := tos(st) + 1]
```

5.2. *The stack cache abstraction*

One of the most interesting issues pertaining to the use of abstraction in specification arose in modeling the process stack. Conceptually, it is desirable to hide the presence of the stack cache so that an application programmer does not have to be concerned with it. In actuality, effect of the stack cache is visible at the macro level in two ways.

For performance reasons, the AAMP does not check memory accesses to determine if the word being referenced lies in the vicinity of the stack cache. As a result, REF and ASSIGN instructions that access the region of memory between the physical TOS and the logical TOS may obtain values different from those held in the stack cache. In practice this does not pose a problem since well behaved applications do not directly access the process stack. Since applications seldom write assembly code for the AAMP (recall that it is designed for use with high order, block structured languages such as Ada), this is primarily a concern for the compiler writers. The second manifestation of the stack cache arises because the AAMP signals an overflow when the physical TOS exceeds the stack limit, rather than when the logical TOS exceeds the stack limit. In effect, the existence of the stack cache allows a few more words to be written to the process stack than should strictly be allowed. While benign, these two manifestations of the stack cache made it difficult to write a precise definition of the AAMP5's behavior without bringing the details of the stack cache into the macro level specification. To avoid this, two adjustments were made to the specification of the macroarchitecture. To reflect the constraint that memory accesses should not be made to the vicinity of the stack cache, we left the effect of such accesses unspecified as shown below.

```
not_stack_cache_address(st:macro_state, base:data_env_ptr) (offset:data_env_addr): bool


data_memory_assign(st     : macro_state,
                   base   : data_env_ptr,
                   offset : (not_stack_cache_address(st, base)),
                   wd     : word): macro_state = st WITH [(dmem)(base)(offset) := wd]

data_memory_ref(st     : macro_state,
                base   : data_env_ptr,
                offset : (not_stack_cache_address(st, base))): word = dmem(st)(base)(offset)
```

The functions `data_memory_ref` and `data_memory_assign` are used exclusively for direct data memory accesses at a given base (data environment pointer) and offset. These functions are constrained by defining them on an appropriate *subtype* of the type of all possible address offsets. The predicate `not_stack_cache_address` is used to define a subtype *dependent*[3] on `st` and `base` to restrict the offset to locations that are not overlaid by the stack cache. The exact definition of the `not_stack_cache_address` predicate is left unspecified in the macroarchitecture since it varies with the specific AAMP microprocessor. One approach would be to be overly conservative and define it as the maximum number of words the stack cache can hold (nine for the AAMP5) above the logical TOS. Another approach is to define it as an abstraction of the micromachine state that contains information about the stack cache as shown below. Such a specification was used in the proofs of correctness relating the micro and macro levels.

```
not_stack_cache_address(ABS(micro_state), base)(offset) =
  (base /= denv(ABS(micro_state)))  OR  offset >= TOS(micro_state)  OR
  (offset < TOS(macro_state) - (SV(macro_state) + TV(macro_state))
```

Using the physical TOS rather than the logical TOS to detect stack overflow was a more difficult problem to resolve. This is dealt with in the proofs by (1) showing the correspondence between the two levels only as long as there is no stack overflow and (2) by proving a correctness statement about the behavior of the stack overflow condition as a separate property of the micromachine. In the AAMP5 a stack overflow can occur only during a procedure call or at the beginning of an instruction during the execution of a special-purpose micro subroutine that performs stack adjustment. The stack overflow correctness is proved as a property of the stack-adjustment micro-routine independent of the instruction being verified.

The issues raised by the stack cache are an excellent illustration of the need for formal, or at least precise, specifications. While the need for speed makes it unlikely that the AAMP will be changed to prevent direct memory accesses in the region of the stack cache, explicitly capturing this constraint provides important documentation for the users of the AAMP, particularly the compiler writers. Using the logical TOS rather than the physical TOS to signal stack overflow is a relatively minor change that may well be incorporated into future AAMPs—this issue had simply never arose since it has such a benign effect. Both issues were brought forward by the requirement to create an abstract, yet precise, specification of the AAMP macroarchitecture.

6. Formal specification of the AAMP5 microarchitecture

The two main technical issues that had to be addressed in specifying the microarchitecture were: (1) the level of detail at which the different parts of the microarchitecture are to be specified and (2) the style of specification to be used to model the design.

One of our objectives was to keep the abstraction level in the microarchitecture specification as close to that used in the official microarchitecture document [24] as possible. The microarchitecture document describes the design approximately at the register-transfer level, with some of the components, such as ALU and the register file, being described as black boxes. The DPU, in which most of the data transformations initiated by the microcode occurs, is specified at the register-transfer level. Since the internal details of LFU and BIU are not needed to verify the correctness of the microcode, the LFU and the BIU are specified more abstractly by formalizing the expected behavior of the interface between those blocks and the DPU. The LFU and BIU provide a set of services for the DPU that can be formalized as a set of safety and liveness properties. Given below are a few of the properties that specify the interface between the DPU and the LFU.

```
RDYinv: AXIOM
   RDY(t) =>
     LET opcode = opcode_at(CENV(t), DP(t), CODE_MEMORY(t))
     IN  EP(t) = EP_ROM(opcode) &
          (has_imm_data(opcode) => IM(t) = first_unit_of_imm_data(t))

CLR_RDY_wait: AXIOM
   NOT RDY(t) & normal_fetching(t) & CLR_RDY?(NCO(t)) =>
      (EXISTS (t1 | t1 > t):
          stays_same(NCO(t))(t, t1)  =>
            stays_same(RDY)(t, t1-1) & RDY(t1) &
               normal_fetching(t1) & DP(t1) = DP(t)+length_of_instruction(opcode) )
```

The first axiom (RDYinv) states a condition on the LFU outputs that the DPU can rely on whenever LFU is ready with a decoded instruction. It states that whenever RDY is true, EP and IM have the appropriate information pertaining to the instruction associated with the program counter DP. The second axiom specifies a possible response of the LFU for a request (CLR_RDY) by the DPU to consume an instruction from the LFU when the LFU is not ready with an instruction. The axiom states that the LFU will eventually present the DPU with a new instruction, provided DPU holds the NC0 line stable with CLR_RDY and that the LFU goes into a state normal_fetching where it presents instructions sequentially.

Formalizing the interface specification was one of the most time-consuming parts of specifying the microarchitecture since it had to be reverse engineered from the LFU/BIU design details and by consulting the design engineers. Nevertheless, abstracting the LFU and BIU provided a natural way of decomposing the verification task. The task of verifying that the LFU/BIU satisfies our interface specification is better-suited for finite-state enumeration tools, such as model-checkers, than theorem provers. To perform such a verification, it is necessary abstract appropriate parts of the LFU-BIU-DPU system to construct a finite-state machine for the model-checker. We have explored [22] this problem for a very simplified model of the AAMP5 using a prototype enhancement to PVS that provides the ability to invoke a model-checker as a decision procedure within PVS.

The DPU is constructed by combining the specifications of the datapath and the microcontroller. The datapath consists of the specification of the register file, the ALU, and the ALU input and output registers. The microcontroller contains the specification of the microsequencer, the stack occupancy logic, and pipelined microinstruction registers.

In most applications of formal logic for the verification of register-transfer-level hardware, the design is modeled as a finite state machine with an implicit clock. The state of the machine consists of the states of the sequential components, such as registers, in the design. The next state function defines the new state value at the next cycle of the implicit clock. The exact style used to specify the state machine implemented by the design can vary. For the AAMP5, we used a *functional* style of specification. In this style, the inputs and outputs of every component are modeled as *signals*, where a signal is a function that maps time, i.e., number of cycles of the implicit clock, to a value of the appropriate type. Every signal that is an output of a component is specified as a function of the signals appearing at the inputs to the component. The signal definitions implicitly specify the connectivity between the components. As an illustration, a specification of the signal denoting the value of the microinstruction register MC0 in the DPU circuit is shown in below.

```
MC0ax: AXIOM MC0(t+1) =  IF DHLD(t) THEN MC0(t)
                         ELSIF abort0(t) THEN null_MC0
                         ELSE romout(t) ENDIF
```

Note that the multiplexor combinational logic that determines the next value for MC0 and the inputs to the combinational logic are implicit in the if-then-else structure in the above definition. The explicit use of a discrete time parameter in the specification helps to express the temporal aspects of interrupt behavior more naturally than in a style which does not use time explicitly.

This style should be contrasted with the *predicative* style [14] that is commonly used in most HOL [15] applications. In the predicative style, every hardware component is specified as a predicate relating the input and output signals of the component and a design is specified as a conjunction of the component predicates, with signals on the internal wires used to connect the components hidden by existential quantification. We decided to use a functional style because proofs based on functional specifications tend to be more automatic than those involving predicative specifications. A proof of correctness for a predicative style of specification usually involves executing a few additional steps at the start of the proof to transform the predicative specification into an equivalent functional style. For a more detailed look at the two styles and their impact on proofs see [19].

7. Verification of the AAMP5

This section describes the verification of the AAMP5. We first present a general correctness principle for microprocessor correctness. We then informally describe the AAMP5 pipeline and then formalize its correctness criterion. Then, we describe the mechanization of the proof of correctness.

7.1. Pipelined microprocessor correctness

Theorem proving approach has been used to verify a number of microprocessor designs since Hunt applied it to verify the FM8501 processor [16]. (For a more comprehensive bibliography of microprocessor verification, see [28].) However, the AAMP5 is significantly more complex, at both the macro and micro-architecture levels, than any other processor for which formal verification has been attempted; it has a large, complex instruction set, multiple data types and addressing modes, and a microcoded, pipelined implementation. Of these, the pipeline and autonomous instruction and data fetching present special challenges. One measure of the complexity of a processor is the size of its implementation. In the case of the AAMP5, this is some 500,000 transistors.

Recently, mechanical verification has been applied to a number of pipelined processors [4, 25, 27, 29, 30]. But, there has not been much work in developing a general framework for microprocessor correctness. In [31], Windley developed such a framework using the notion of generic interpreters, but it is not applicable to pipelined processors. We derived the AAMP5 correctness criterion using the general framework developed by David Cyrluk [9] for proving correspondence between state machines in PVS which has been shown to be applicable to pipelined processors.

Formal microprocessor verification typically involves specifying the processor as a machine that interprets the instructions in the instruction set at two levels—*macro* and *micro*. The macromachine gives the programmer's view of the processor, hiding internal details such as pipelining. The micromachine describes the processor at a lower level, such as the architecture or register-transfer level.

The general correctness criterion for a microprocessor is expressed, as shown in figure 5, as a relationship between the execution traces of the macro and micromachines. In the figure, `E` and `next_macro_state` denote the micromachine and macromachine state

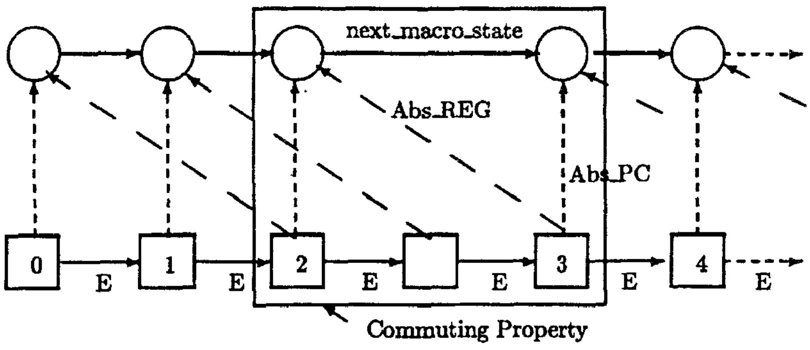


Figure 5. Pipelined microprocessor correctness.

transition functions, respectively. The objective of the correctness criterion is to ensure that the micromachine does not introduce any behaviors not allowed by the macromachine. Since the micromachine usually takes a number of cycles to execute an instruction that is viewed atomically at the macro level, the two machine traces are required to correspond only at selected points (called *visible* states) in the micromachine trace. For example, one possible definition of a visible state is one in which an instruction gets completed and a new instruction begins. A visible state is indicated in figure 5 by placing the number of the current instruction in that state. Since the macrostate is an abstraction of microstate, it is necessary to define an *abstraction* function ABS (in figure 5, for now, read the vertical arrow labeled Abs_PC as ABS and ignore the Abs_Reg arrow) that returns the macrostate corresponding to the state of the micromachine for each visible state.

Given the notions of ABS and visible state, the correctness criterion described by figure 5 can be stated as follows[4]: "Every micromachine trace starting with an initial state (S0) that abstracts to a macrostate (s0) must be abstractable to the macromachine trace starting with the initial state s0." To prove such a correspondence between two infinite traces, it is sufficient to prove the *commuting* property (stated formally below) that captures the correspondence between the traces for an arbitrary step of the macromachine, i.e., between two arbitrary but successive visible states. This property is stated using existential quantification since the distance between visible states can vary. In the formal statement shown below, ABS and other functions that take a time argument *t* implicitly operate on the micromachine state at *t*. Hence, the micromachine transition function *E* is implicit in the increment of time.

```
commuting_property: LEMMA (FORALL (t: time):
     visible_state(t) => EXISTS (tp: time | tp > t):
             stays_low(t+1, tp-1)(visible_state) &
                 visible_state(tp) & ABS(tp) = next_macro_state(ABS(t))
```

The commuting property described in the previous paragraph can also be used for a pipelined processor if appropriate adjustments are made. First, in a pipelined processor, a new instruction enters the pipeline before the results of the previous uncompleted instructions are at their destination. This can be handled by allowing ABS to peek into future, possibly nonvisible, microstates (e.g., Abs_REG shown in figure 5) to get correct values for some of the macrostate components. Typically, the values for all but the program counter (Abs_PC in figure 5) must be obtained from the future state. We refer to the distance into the future from the current visible state where the information is to be obtained as the *latency* for the abstraction function.

Second, some pipelined processors, typically RISC processors, do not inhibit all kinds of pipeline hazards. For example, they allow instructions following a conditional branch to execute, assuming the branch would not be taken, leaving the burden of generating correct code to the compilers. One way to handle this (as shown in [27]) is to either restrict the set of programs for which the two machines are related or expose some of the effects of pipelining in the macromachine. Further adjustment would be needed if pipelining allows out-of-order instruction completion.

Although the AAMP5 was designed to be object-code compatible with its nonpipelined CISC-like predecessor, it's design employs a number of features commonly found in RISC processors. The AAMP5 pipeline exhibits both control and data hazards, but does not employ out-of-order execution. We handled the effect of latency on pipelined execution of instruction by suitably adjusting the ABS definition. Although the effect of pipelining is completely hidden from the programmer, the AAMP5 exposes the effect of stack caching. In the AAMP5 verification, the lack of transparency of stack caching had to be handled similar to the way in which exposing the effect of pipelining would be handled. The main challenges we faced in formalizing the correctness criterion for the AAMP5 and mechanizing its proof were the following.

First, executing a large, complex and irregular instruction set in a pipeline creates a rather large number of conditions in which the pipeline is interrupted and extended. This irregularity makes characterizing a suitable visible state predicate hard and increases the number of special cases to be considered in the proof. Second, the caching of process stack in internal registers introduces a gap between the memory models at the micro and macro levels that is nontrivial to bridge.

Third, in the AAMP5 instruction fetches and data transfers happen autonomously with respect to the interpretation of instructions in the pipeline. This design characteristic makes it hard to choose a reference point for characterizing the visible state since the distance between two consecutive visible states is indefinite and varying. If the definition of the visible state is not engineered carefully, the abstraction can become complicated. Burch and Dill [4] have proposed the idea of using a sequence of NOPs equal to the latency of the instruction to flush out the states of the desired internal registers to eliminate the need to explicitly define an abstraction function in some microprocessor verifications. This idea cannot applied for the AAMP5 because of the indefinite distance between visible states and also because of the nontrivial abstraction gap between the micro and macro levels.

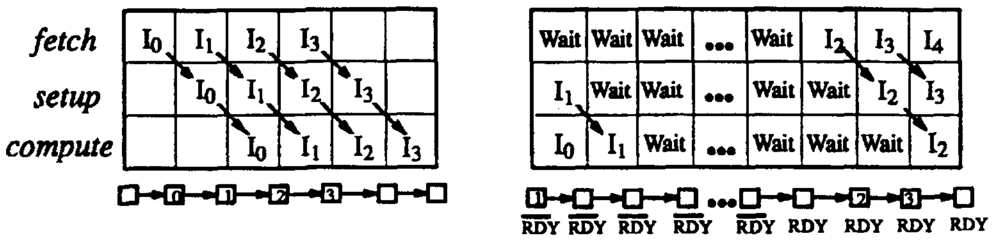
7.2. *The AAMP5 pipeline*

The reader is advised to refer to figure 2 in Section 3.2.1 while reading this section. The AAMP5 executes its instructions in a three-stage pipeline: *fetch*, *setup*, and *compute*. The fetch stage consists of the DPU activities that occur prior to the entry of the first microinstruction of an AAMP5 instruction into MC0. This includes reading the microinstruction at EP from the ROM and checking if the instruction requires a stack adjustment, and if so, performing the stack adjustment. The setup stage consists of the activities controlled while the microinstruction is in MC0, which includes computing the new stack cache occupancy state and reading the register file to determine the sources for the ALU input registers R and W. The compute stage consists of the activities determined by the part of the microinstruction that moves from MC0 to MC1. It includes computing the results of the ALU, checking if there will be delayed jump and initiating writing of the ALU results. If a delayed jump is detected (as a function of the ALU outputs) in the second stage, then both MC0 and MC1 are cleared to abort the operations normally performed by the MC0 and MC1 microinstructions and the jump-address field of MC1 is passed onto MC2, which will be used as the target microaddress of a potential delayed jump. A delayed jump is typically used to implement exceptions such as arithmetic overflow. Note that the register MC2 is used only for delayed jump execution and does not add a stage under normal instruction execution.

7.2.1. Normal pipeline operation. The DPU logic is designed so that the DPC0 register always contains the program counter value associated with the macroinstruction that is being interpreted in MC0. A similar correspondence holds between DPC1 and MC1 as well as DPC2 and MC2. We use the macroinstruction associated with DPC0 as the reference point and refer to it as the *current* instruction. Under *normal* pipeline operation, i.e., when AAMP5 instructions are being completed at the rate of one per cycle, the DPU performs the setup stage of the current instruction along with the compute stage of the previous instruction and the fetch stage of the next instruction.

The execution trace of the micromachine under normal pipeline operation is shown in figure 6(a). A state (square) in the execution trace (shown at the bottom of figure 6) with a number n in it denotes the visible state where instruction n is the current instruction. The column above a state transition arrow shows the instructions in the pipeline and the stages that are being performed in a given cycle.

7.2.2. Pipeline stalling and delayed jumps. One of the challenges posed by the AAPM5, from the point of view of verification, is that its CISC instruction set is not as regular as a



a) Normal Pipeline Operation b) Stalling Due to LFU Not Being Ready

Figure 6. Pipeline Execution Traces.

RISC instruction set for pipelined execution. As a result, the number of conditions under which the normal operation of the pipeline can be interrupted and/or extended is more than one would normally find in a RISC machine. Not only does this aspect make the formalization of a visible state harder, but also complicates verification because the proof of correctness under these conditions has to be handled in special ways. When the AAMP5 pipeline is at a visible state, the pipeline operates normally (i.e., the distance to the next visible state is one cycle) only if the following conditions hold: (1) The current instruction, i.e., the one in the setup stage in the pipeline, is implemented by a single microinstruction. (2) The previous instruction, i.e., the one in the compute stage, (a) does not cause a delayed jump (nextDJMP, the value assigned to DJMP in the next cycle, is false), and (b) will not cause a data hold (DHLD is false). (3) The LFU is ready with the next instruction (RDY is true). (4) The next instruction will not need a stack adjustment (SADJcondn is false).

If any of the conditions listed do not hold, the distance to the next visible state will be more than one cycle and its value will depend on which of the conditions in the list are violated. We classify these situations in which the next visible state is more than one cycle away from the present one as follows: The behavior resulting from violations of conditions 2(b), 3, or 4 is referred to as pipeline *stalling*; in these situations, the micromachine performs a computation that has no visible effect on the macrostate. Violations of the other two cases (1, 2(a)) are referred to as pipeline *extension*. In a pipeline extension, the micromachine components that affect the macrostate are updated incrementally in two or more cycles. In all these situations, the compute stage of the previous incomplete instruction is always completed in the following cycle.

For example, figure 6(b) shows the trace when the LFU is not ready with the next instruction. In this case, a jump is made to a special microroutine (MAP_WAIT) that implements a busy wait loop which is exited only when RDY becomes true again. The distance to the next visible state can be arbitrarily large depending on when RDY becomes true again.

7.2.3. The AAMP5 pipeline correctness.

The correctness of the AAMP5 pipeline is formalized by appropriately instantiating the general commuting property formalized in Section 7.1. For this, we first need to define ABS and `visible_state` for the AAMP5.

The main issue to be considered in characterizing the visible state is which of the macroinstructions in the pipeline and what point of its execution is to be used as the reference point to relate the microstate to the macrostate. In principle, one can use any of the instructions in the pipeline, but the choice influences how complex the definition of the abstraction function will be. In most pipeline verifications, the instruction in the fetch stage is used as a reference point. This choice, however, is not suitable in the present case because variations in the latency of the pipeline due to pipeline stalling will be embedded in the definition of the abstraction function. For automating proofs, it is advisable to choose the reference point so that abstraction function need only look a fixed distance into the future.

So, we define a visible state as one in which the first microinstruction of a new AAMP5 instruction is in MC0 and is guaranteed to advance to MC1 in the next cycle. That is, the left-over effects of the previous instruction are guaranteed to be completed in the next cycle. Under this characterization of the visible state, the latency for the ABS function is exactly one cycle. During normal pipeline operation, the distance between consecutive visible states is exactly one cycle. A formal characterization of a visible state is given below.

```
visible_state_with(op: opcodes)(t): bool =
  MC0(t) = ROM(zero_extend[8](11)(EP_ROM(op)(t))) &
    & NOT(DHLD(t)) & NOT(nextDJMP(t)) & entry_cond_met(op)(t)

visible_state(t): bool =
    EXISTS (op: opcodes): visible_state_with(op)(t)
```

`Visible_state_with(op)(t)` *defines the condition that must be satisfied for a* micromachine state to be visible with the opcode of the current instruction being `op`. The first condition in the conjunction ensures that MC0 contains the first microinstruction of the microcode corresponding to `op`. The remaining conditions ensure that the previous instruction will complete in the next cycle: DHLD being false ensures that the previous instruction will not be held due to data memory transfer; `nextDJMP(t)` (which is the value of the DJMP register one cycle later) being false ensures the previous instruction will not cause a delayed jump; `entry_cond_met` ensures that the entry conditions of the instruction in MC0 are met by the stack occupancy state determined by the SV1 and TV1 registers.

The abstraction function gets the value for every field in the macrostate record, except `TOS` and `dmem`, directly from the state of a corresponding component in the micromachine. The definition of the abstraction function is divided into a separate function for each of the components in the macrostate. Given a time `t`, an abstraction function returns a component of the macrostate corresponding to the microstate at `t`. The abstraction values for `dmem` and the logical `TOS` are shown below.

```
stack_cache_size(t): nat = bv2nat(SV1(t)) + bv2nat(TV1(t))

ABStos(t): word = TOS(t+1) - stack_cache_size(t+1)

i: VAR nat
ith_top_of_scache: [time -> [nat -> word]]
ith_top_of_scache: AXIOM i < stack_cache_size(t)  =>
  ith_top_of_scache(t)(i) = IF i < bv2nat(TV1(t)) THEN IF i = 0    THEN T0(t)
                                                         ELSIF i = 1 THEN T1(t)
                                                         ELSE T2(t) ENDIF
                            ELSE REG(t)(stack_cache_size(t)-i-1) ENDIF

ABSdmem(t)(dptr: data_env_ptr)(daddr: word): word =
   IF bv2nat(daddr) < bv2nat(TOS(t+1)) &
        bv2nat(daddr) >= bv2nat(TOS(t+1)) - stack_cache_size(t+1)
   THEN LET offset =  stack_cache_size(t+1) - (bv2nat(TOS(t+1)) - bv2nat(daddr))
        IN ith_top_of_scache(t+1)(offset)
   ELSE DATA_MEMORY(t+1)(dptr o daddr) ENDIF
```

Since the elements of the stack cache are viewed as being in the data memory at the macro level, the macro TOS is smaller than the micro TOS by the size of the stack cache, i.e., (SV1 + TV1). The locations of `dmem` correspond to the locations in DATA_MEMORY except for those that range from the micromachine (TOS $-$ 1) to (TOS $-$ (SV1 + TV1)) held in the stack cache. A formalization of the commuting property for the AAMP5 is given below.

```
commuting_property: LEMMA
  visible_state(t) & proper_instrns_in_pipe(t) & no_logical_stack_overflow(t)
    => EXISTS (tp: time | tp > t):
              stays_low(t+1, tp-1)(visible_state) &
                  visible_state(tp) & ABS(tp) = next_macro_state(ABS(t))
```

The commuting property has two additional preconditions when compared to the general commuting diagram of Section 7.1 to handle the consequences, described in Section 5.2, of the hiding of the stack cache from the macro level. `Proper_instrns_in_pipe(t)` ensures that the memory address operands (if any) of every macroinstruction in the pipeline are not within the area overlaid by the stack cache; `next_macro_state` is left undefined if this predicate were false. `No_logical_stack_overflow(t)` ensures that the logical process stack would not overflow as a result of executing the current instruction.

7.3. *Mechanizing the proof of correctness*

A common strategy used to prove the commuting property is to start from an arbitrary visible state at the lower left-hand corner of the box representing the community property in figure 5, traverse the two possible paths to the top right-hand corner of the box, then check whether the two states resulting from the traversals are equivalent. Traversing the top path, which entails applying the abstraction function followed by "running" the macromachine one step, results in the *expected* state. Traversing the bottom path, which consists of running the micromachine a multiple number of cycles followed by an application of the abstraction function, gives the *actual* state.

In a proof, running the micro and the macromachines can be accomplished by *rewriting*, which consists of unfolding every occurrence of a defined function with its definition until all the functions have been simplified to primitive expressions. Checking whether the expected and actual states are equivalent is essentially a problem of checking the equivalence of boolean or bit-vector expressions since the state of most of the machine components is modeled as either boolean or bit-vector types. As discussed in Section 7.2, the number of cycles that the micromachine needs to be run depends on the type of current instruction and whether one or more of the stalling conditions apply. Hence, a proof typically consists of a case analysis on these conditions.

Our main objective in deciding on a suitable approach for mechanization of the proof was two-fold. First, the verification method had to be automatic enough so that the Collins engineers could perform proofs of instructions after proofs of a few representative instructions were demonstrated to them. Second, the verification method should permit incremental and partial verification of instructions. The importance of automation cannot be overemphasized since it is a pre-requisite for transfer of this technology to practice. We also wanted the ability to perform partial verification since that is a good way of getting early feedback from the verification process.

In the present case, one of the main reasons a complete automation of the commuting property was not possible is that the number of cycles for completing the current instruction can be indefinite due to pipeline stalling caused by memory wait states and stack adjustments. One way to facilitate automation, is to transform the verification problem into one of proving a set of properties each of which relates a pair of states of the micromachine that are a fixed and finite distance apart in time.

As shown in [8], for register-transfer level designs specified in the functional style described in Section 6 such "finite distance" properties can be automatically proved using a *core hardware strategy*. The hardware strategy consists of repeatedly performing the following sequence of proof steps: (1) rewriting the property using the design specification, (2) lifting the boolean expression in if-then-else structures to the top-level, and (3) simplification using PVS's BDD and other decision procedures. The special-purpose decision procedures for microprocessors proposed in [4, 7] are efficient methods for proving such properties under certain restrictions. Our hardware strategy is more powerful than the procedures in [4, 7] because our strategy uses the arithmetic and equality decision procedures of PVS.

To accomplish such a transformation of the verification problem, we decomposed the proof of the commuting property into three parts.

1. A part that reasons exclusively about the stalling behavior of the pipeline at the micromachine level.
2. A part that reasons about the instruction correctness in the absence of stalling at the micromachine level.
3. A part that combines the first two parts along with the task of relating the micromachine to the macromachine by applying the abstraction function.

This decomposition was natural since pipeline stalling is implemented by a set of special-purpose microroutines that are invoked as a subroutine during an instruction execution. The correctness of pipeline stalling was characterized by a set of formulas called *general verification conditions* since they are common to all instructions. The correctness of instructions in the absence of stalling were formalized by a separate set called *instruction-specific* verification conditions.

This decomposition offered a number of advantages. First, it satisfied our goal of incremental verification since the proof of each part can be carried out in parallel once the verification conditions are formulated. It enhances the chance of early error detection because microcode analysis is involved only in the first two parts. Second, it satisfied our goal of effective automation. Proofs in the first and the third parts were done using the core strategy with a few augmentations as described below. These proofs are automatic enough for a nonexpert to perform. Proofs in the general verification conditions involved combining the core strategy with induction. The two kinds of verification conditions are described below.

7.3.1. General verification conditions. A general verification condition states a property of the following form: "if the pipeline is in a stalling state then it will eventually progress to an appropriate non-stalling state without affecting the externally visible parts of the micromachine state." Since the general verification conditions are not in the form of a property that relates states that are a finite distance apart, their proof is not as automatic as that of the instruction-specific verification conditions. The proof involves induction on the length of the time taken by the memory to respond or the size of the stack adjustment needed and requires involvement by an expert. However, we only needed to formulate about half a dozen general verification conditions to characterize the three stalling situations discussed in Section 7.2.2. Since the general verification conditions are common to all instructions, their proof needed to be done once for all instructions.

*7.3.2. **Instruction-specific verification conditions.*** In formulating the instruction-specific verification conditions, it was necessary to apply an additional decomposition step to handle the fact that even in the absence of pipeline stalling the length of execution of an instruction, although definite, can vary depending on the type of the current instruction and whether an instruction causes an exception or user-recoverable error. One way to handle the variation in the length of the execution time for different instructions is to define an "oracle" function that gives, a priori, the expected execution time for all instructions under different situations. Another possible approach is to partition the proof by hand into different instructions and conditions according to the execution times. We took the second approach since it was more amenable to incremental and partial verification.

For example, shown below are two of the instruction-specific verification conditions for the ADD instruction under the condition that addition does not overflow. The overflow situation has a separate set of verification conditions, as does every situation that requires a distinct number of cycles to complete the ADD instruction execution. In the absence of an exception, the latency of ADD is 2 cycles.

```
TO_correctness: LEMMA
   visible_state_with(ADD)(t) =>
      TO(t+2) = (sign_extend[16](48)(ith_top_of_scache(t+1)(1))

next_macro_instrn_entry: LEMMA
   visible_state_with(ADD)(t) & NOT Soverflow(t + 1) =>
       DPC(t+1) = DPC(t) + length_of_instruction(opcode_at_DPC(t)) &
          visible_state_with(next_opcode_at_DPC(t))(t+1) )
```

The first formula expresses constraints on the T0 register, which holds the top element of the process stack. The second formula ensures that the expected next instruction moves into the pipeline. The verification conditions state the effect on the state of each of the microma-chine components relevant to the macrostate by the execution of an instruction, in this case ADD. It is important to note that the expected value for the state of a component is expressed only in terms of the initial values of the states of the macro-level relevant components. Thus, although the verification conditions do not relate the microstate to the macrostate, they are, in effect, making assertions about the intended changes to the macrostate.

In our methodology, we envision the design engineer or the nonexpert's primary involvement in the verification phase to be to formulate and prove the instruction-specific verification conditions for the instructions in the instruction set. The design engineer would be able to formulate these conditions without much difficulty because these conditions are stated in terms of a model that the design engineer can readily understand. At the same time, the guidelines for formulating these conditions ensure the instruction behavior is completely covered.

Our core hardware strategy is directly applicable to the instruction-specific verification conditions since they are in the form of a "finite distance" property. Since the AAMP5 datapath involves a significant amount of bit-vector manipulation outside the ALU, we had to augment our core hardware strategy with a capability to check equivalence on bit-vector expressions involving a set of commonly used bit-vector operations. The bit-vector equivalence checking was performed by formulating a set of rewrite rules that is automatically applied by our strategy to normalize a bit-vector expressions into a cannonical form in most cases. Even when the hardware strategy is unable to automatically prove a verification condition, it reduces the proof to one or more goals of equality on two bit-vector expressions. The bit-vector expressions in these unproved goals are simplified enough that the designer can inspect them to check if the expressions are really not equivalent or equivalent but the bit-vector normalization was not able to reduce them to a cannonical form. In the latter case, the designer has to manually invoke a special set of pre-defined rewrite rules to finish the proof. The former situation implies an error in the design or the specification that needs the designer's attention.

The proof of combining the verification conditions to show the commuting property was also done using the core hardware strategy with a slight improvisation to suit the specific details involved in the definition of the abstraction function. By embedding the improvisation needed inside another special-purpose strategy, we believe this part can also be performed by a nonexpert.

8. Conclusions

The central goal of the AAMP5 project was to explore the technical feasibility of formally specifying and verifying a complex commercial microprocessor and develop a microprocessor verification methodology that could be used by practicing engineers who are not formal verification experts. This section discusses the lessons learned on this project and their implications for the industrial use of formal methods.

Feasibility. A much larger fraction of the AAMP instruction set was specified than originally planned, with 108 of the AAMP's 209 instructions completed. The portion completed is actually greater than this, since many of the instructions specified are representative of an entire family of instructions. All of the micro-architecture needed for formal verification of the microcode was formally specified. At this time, eleven instructions have been proven correct in the absence of interrupts. The methodology and the automatic strategy was applied by one of the Collins engineers to partially verify an additional instruction.

Although the number of instructions verified was small, the verification methodology and the proof strategies developed for the effort are applicable to any instruction whose latency (ignoring the time consumed due to stalling) can be expressed as a function of its opcode attributes alone or as a simple condition, such as exceptions, overflow, or test for zero, etc., on its operands. This class covers about 70 percent of the AAMP5 instructions. Formalizing the verification conditions for a new instruction in this class and verifying it completely can typically be done in half a day to a day depending on how close the new instruction is to an already verified instruction. Rerunning the complete proof of a 3-cycle instruction takes about half an hour on a SPARC 10 machine. If the goal is to verify a large number of instructions completely, then a more effective approach than the incremental approach we adopted is to construct a single strategy that eliminates the intermediate step of formulating verification conditions specific to each instruction. For this combined approach to be successful for a large design, it is imperative to have an effective decision procedure for bit-vectors and a very efficient rewriting engine.

Our strategy is not applicable to instructions, such as multiply and divide, whose latency is data dependent in a complex fashion. Verifying such instructions require sophisticated use of induction and invariants. We did not verify the behavior of interrupts although we specified the processor reset which is treated as an interrupt. The correctness of an interrupt can also be decomposed into a set of verification conditions similar in form to the instruction-specific verification conditions and, hence, our proof strategies are applicable for them. Finally, our verification was based on the assumption that the LFU and the BIU satisfied a set of postulated properties. While these protocol-like properties can be verified in PVS using induction, model-checking tools are better suited for such verification. In [22], we show how such properties can be effectively proved using a combination of theorem proving and model-checking techniques.

Cost. The cost of developing and validating the macro and micro-architecture specifications and developing the proofs of correctness were significant, but many of these expenses have to be attributed to the exploratory nature of the project. Reuse of specifications, such as the bit vector theories, proof strategies, and expertise should greatly reduce these costs in the future. A large portion of the time spent on the correctness proofs was invested in the development of reusable proof strategies rather than just proving the correctness of the core set of instructions. Even so, we feel that more effective and efficient automation of lower level proofs involving rewriting and bit-vectors are needed to make formal verification cost-effective. For example, some of the decompositions that we used to facilitate our proofs would not have been necessary if the hardware strategy could be implemented more efficiently. Toward this end, it would be interesting to explore integrating special purpose decision procedures, such as [4], into PVS.

Benefits. While it is not hard to convince engineers that formal methods subject a design to a more thorough analysis than traditional methods can, these benefits must also be evaluated relative to their cost. Until technological advances reduce the cost of formal verification significantly below the present level, it will be necessary to apply formal verification prudently. One way to maximize the benefits is to apply it to well-understood problem domains, such as microprocessor designs. The ability to perform partial verification is crucial since that facilitates "looking in corners" and consideration of unusual cases early and independently. The incremental strategy was used to our advantage on this project. We planned the project so that we would get useful feedback from the formal analysis as quickly and as early as possible. The specification phase was designed to cover the architecture as completely as possible beyond the minimum required and the infrastructure for performing the proof of correctness was set up to construct proofs incrementally and as automatically as possible.

Two errors were found while constructing the macroarchitecture specification. Both errors were found while trying to formally specify the behavior of the AAMP under unusual circumstances that were not clearly specified in the AAMP2 Reference Manual, prompting a team member to examine the microcode in the AAMP5. The first was a logic error that allowed the top of stack register (TOS) to wrap around a data environment instead of raising a stack overflow. To result in a failure, this error required the very unlikely combination of an unusual system configuration, an improperly sized stack, and a specific sequence of instructions. The second error was made precisely because the reference manual was unclear on how the AAMP should update the local environment register (LENV) when a procedure call caused a stack overflow. This was implemented by setting the LENV to its "overflow" value, while the correct behavior was to leave the LENV unchanged. Both errors were unique to the AAMP5 and corrected before first fabrication.

The proof of correctness phase revealed a number of errors. In the initial stages, most of the errors found were errors in our micromachine specification, although the specification had been subjected to formal inspections. Two of the errors revealed during the proof were actual errors in the microcode. Both of these errors were known to the AAMP5 designers but were left in the microcode supplied to SRI for verification without divulging any information about the errors. One of the microcode errors was in the microcode for the REFBXU instruction, which loads data from memory to the top of process stack, while the other was in the ASNDXI instruction, which stores data in the process stack to a location in memory. While the second error was deliberately planted by Collins engineers in the microcode, the first one had escaped the the microcode traditional walk-throughs and simulations. The two errors were similar in that they occurred in the calculation of the memory address and are hard to spot during microcode walk-throughs because of the pipelining of microinstructions. More details are given in [28].

Technology transfer. It is very difficult to inject new methodologies into an industrial setting since one of the ways industry remains competitive is to use tried and tested approaches within a well understood problem domain. Despite their name, formal methods provide remarkably little methodology to guide their use in a new setting. Given this, it seems prudent to plan for costs to be high the first time around and to expect most of the benefits to appear on subsequent projects of a similar nature.

We did not feel that it was particularly difficult for the engineers at Collins to learn to use either the PVS language or the theorem prover. In fact, it was much easier for them to apply formal methods than it was for the formal methods experts to become knowledgeable about the AAMP5. The real problem was not how to use PVS, but how to build a precise mathematical model of our own microprocessor. Even so, widespread acceptance of a general purpose specification language such as PVS or Z [5] by practicing engineers is likely to be an uphill battle. A more productive approach may be to develop specialized notations or models that fit a specific problem domain and that can automatically be translated into an underlying formalism such as PVS. This would allow the domain experts to work in a familiar and natural notation while a small group of formal methods experts (and tools) check their work for consistency and completeness.

Validation of formal specifications is essential to have confidence in the correctness proofs. We found inspections worked well with formal specifications, were quite inexpensive, and provided a natural vehicle for training. Maximizing the independence of the teams producing the specifications greatly increased our confidence in both the proofs and the specifications. When combined with proofs of correctness, this is a very powerful validation technique that should not be overlooked. Other forms of validation that could have been used more extensively in this project include early proof of the type correctness conditions generated by PVS and proving expected properties, or putative theorems, of the specification. Even our limited experience with proving putative theorems suggests that this is a useful validation technique.

It is our belief that the groundwork performed on this project will greatly lower the cost of specifying and verifying another member of the AAMP family. To test this hypothesis we have begun a new project to use the methodology developed to verify another member of the AAMP family in its entirety.

Acknowledgments

The authors thank Rick Butler and Paul Miner of NASA Langley Research Center for their support and refinement of the bit-vectors library, Sam Owre and N. Shankar of SRI International for their prompt and dedicated work at maintaining and enhancing PVS which greatly contributed to the success of this project, and Al Mass and Dave Greve of Rockwell International for their many hours on this project. We also thank John Rushby of SRI International and John Gee, Dave Hardin, Doug Hiratzka, Ray Kamin, Charlie Kress, Norb Hemesath, Steve Maher, Jeff Russell, and Roger Shultz of Rockwell for their assistance and support.

Notes

1. A theory is a logical unit of specification in PVS that consists of a collection definitions, and possibly statements of properties to be proved.
2. Although formal analysis of the interrupt behavior of the AAMP5 was not included in this effort, the basic framework developed is adequate to allow it to be easily added.
3. In PVS a subtype is declared by enclosing the predicate defining the subtype in parentheses. Dependent subtypes are an important PVS feature allowing a type component to depend on earlier components.
4. We also need to ensure that processor resetting will always lead to an initial state in which the correspondence required by ABS holds.

References

1. D. Best, C. Kress, N. Mykris, J. Russel, and W. Smith, "An advanced-architecture cmos/sos microprocessor," *IEEE Micro*, pp. 11–26, August 1982.
2. R.S. Boyer and J.S. Moore, *A Computational Logic*, Academic Press, New York, NY, 1979.
3. S. Brock and C. George, *The RAISE Method Manual*, Computer Resources International A/S, 1990.
4. J.R. Burch and D.L. Dill, "Automatic verification of pipelined microprocessor control," in David Dill (Ed.), *Computer-Aided Verification, CAV'94*, Vol. 818 of Lecture Notes in Computer Science, pp. 68–80, Stanford, CA, June 1994, Springer-Verlag.
5. A. Burns, J. McDermid, and J. Dobson, "On the meaning of safety and security," *Computer Journal*, Vol. 35, No. 1, pp. 3–15, 1992.

6. Ricky W. Butler, "NASA Langley's research program in formal methods," in *COMPASS'91, Proceedings of the Sixth Annual Conference on Computer Assurance*, Gaithersburg, MD, June 1991, pp. 157–162, IEEE Washington Section.
7. D. Cyrluk and P. Narendran, "Ground temporal logic—a logic for hardware verification," in David Dill (Ed.), *Computer-Aided Verification, CAV'94*, Vol. 818 of Lecture Notes in Computer Science, pp. 247–259, Stanford, CA, June 1994, Springer-Verlag.
8. D. Cyrluck, S. Rajan, N. Shankar, and M.K. Srivas, "Effective theorem proving for hardware verification," in Kumar and Kropf [18], pp. 203–222.
9. Cyrluk, "Microprocessor verification in PVS: A methodology and simple example," Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
10. Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell, "Formal design and verification of a reliable computing platform for real-time control," NASA Technical Memorandum 102716, NASA Langley Research Center, Hampton, VA, October 1990.
11. M. Fagan, "Advances in software inspections," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, pp. 744–751, 1986.
12. S. Gerhart, M. Bouler, K. Greene, D. Jamsek, T. Ralston, and D. Russinoff, "Formal methods transition study final report," Technical Report STP-FT-322-91, Microelectronics and Computer Technology Corporation, Austin, Texas, August 1991.
13. James Glanz, "Mathematical logic flushes out the bugs in chip designs," *Science*, Vol. 267, pp. 332–333, 1995.
14. M. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," Technical Report 77, University of Cambridge Computer Laboratory, September 1985.
15. M.J.C. Gordon and T.F. Melham (Eds.), *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, Cambridge University Press, Cambridge, UK, 1993.
16. Warren A. Hunt, Jr., *FM8501: A Verified Microprocessor*, Vol. 795 of Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, 1994.
17. Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International Series in Computer Science, Prentice Hall, Hemel Hempstead, UK, second edition, 1990.
18. Ramayya Kumar and Thomas Kropf (Eds.), *Theorem Provers in Circuit Design (TPCD'94)*, Vol. 910 of Lecture Notes in Computer Science, Bad Herrenalb, Germany, September 1994, Springer-Verlag.
19. Mandayam Srivas et al., "Hardware verification using PVS: A tutorial," Technical Report SRI-CSL-95-13, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995 (Forthcoming).
20. Steven P. Miller and Mandayam Srivas, "Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods," in *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995, pp. 2–16, IEEE Computer Society.
21. S. Owre, N. Shankar, and J.M. Rushby, *The PVS Specification Language*, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993 (A new edition for PVS Version 2 is expected in early 1995).
22. S. Rajan, N. Shankar, and M.K. Srivas, "An integration of model-checking with automated proof checking," in Pierre Wolper (Ed.), *Computer-Aided Verification, Cag'95*, Vol. 939 of Lecture Note in Computer Science, pp. 84–97, Liege, Belgium, June 1995, Springer-Verlag.
23. *AAMP2 Advanced Architecture Microprocessor II Reference Manual*, Rockwell International, Collins Commercial Avionics, Rockwell International Corporation, Cedar Rapids, Iowa 52498, February 1990.
24. *AAMP5 Microarchitecture (Unreleased Document)*, Rockwell International Processor and Software Technology Department, Advanced Technology and Engineering, Collins Commercial Avionics, Rockwell International Corporation, Cedar Rapids, Iowa 52498, February 1993.
25. James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning, "Using transformations and verification in circuit design," *Formal Methods in System Design*, Vol. 4, No. 1, pp. 181–210, 1994.
26. N Shankar, S. Owre, and J.M. Rushby, *The PVS Proof Checker: A Refrence Manual*, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993 (A new edition for PVS Version 2 is expected in early 1995).
27. Mandayam Srivas and Mark Bickford, "Formal verification of a pipelined microprocessor," *IEEE Software*, Vol. 7, No. 5, pp. 52–64, 1990.
28. Mandayam Srivas and Steven P. Miller, "Formal verification of a commercial microprocessor," Technical Report SRI-CSL-95-4, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1995.

29. S. Tahar and R. Kumar, "Implementing a methodology for formally verifying risc processors in hol," in Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.), *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG'93)*, Number 780 in Lecture Notes in Computer Science, pp. 281–294, Vancouver, Canada, August 1993, Springer-Verlag.
30. Phillip J. Windley and Michael L. Coe, "A correctness model for pipelined microprocessors," in Kumar and Kropf [18], pp. 33–51.
31. P.J. Windley, "Formal modeling and verification of microprocessors," *IEEE Transactions on Computers*, Vol. 44, No. 1, 1995.